Advanced Components in the Variable Precision Floating-Point Library

Xiaojun Wang, Sherman Braganza, Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA, USA
{xjwang,sbraganz,mel}@ece.neu.edu

Abstract

Optimal reconfigurable hardware implementations may require the use of arbitrary floating-point formats that do not necessarily conform to IEEE specified sizes. We have previously presented a variable precision floating-point library for use with reconfigurable hardware. We recently added three advanced components: floating-point division, floating-point square root and floating-point accumulation to our library. These advanced components use algorithms that are well suited to FPGA implementations and exhibit a good tradeoff between area, latency and throughput. The floating-point format of our library is both general and flexible. All IEEE formats, including 64-bit double-precision format, are a subset of our format. All previously published floating-point formats for reconfigurable hardware are a subset of our format as well. The generic floating-point format supported by all of our library components makes it easy and convenient to create a pipelined, custom datapath with optimal bitwidth for each operation. Our library can be used to achieve more parallelism and less power dissipation than adhering to a standard format. To further increase parallelism and reduce power dissipation, our library also supports hybrid fixed and floating-point operations in the same design. The division and square root designs are based on table lookup and Taylor series expansion, and make use of memories and multipliers embedded on the FPGA chip. The iterative accumulator utilizes the library addition module as well as buffering and control logic to achieve performance similar to that of the addition by itself. They are all fully pipelined designs with clock speed comparable to that of other library components to aid the designer in implementing fast, complex, pipelined designs.

1 Introduction

One of the applications of reconfigurable hardware is to accelerate algorithms initially implemented in software and run on a general purpose processor. The prime candidates are those algorithms that are highly regular and parallel, such as image and signal processing applications. To achieve acceleration, custom datapaths are built in reconfigurable hardware using either fixed-point or floating-point arithmetic. In either case, optimal bitwidths of the signals in the datapath are application specific, and depend on the values they carry. To increase potential parallelism and to reduce the power dissipation of the circuit, we wish to minimize the bitwidth of each signal. Hence, it is important to have fine-grained control over the datapath bitwidths throughout the whole design process. Arbitrary fixed-point formats are not difficult to implement and are in common use. However, due to the inherent complexity of floating-point representation, arbitrary floating-point formats are much harder to implement though they are no less desirable.

We have implemented a variable precision floating-point library [1]. Our library supports general flexible formats which can be parameterized to accept any bitwidth operand including the IEEE standard formats. Most earlier work only supports either IEEE formats or the authors' non-general application-specific formats. Each component in our library has a READY signal and a DONE signal to support pipelining. All the hardware modules, including the three advanced components presented in this paper, are fully pipelined. Our library includes components to convert between fixed-point and floating-point format, which allows the user to implement the optimal hybrid datapath comprised of both fixed-point and floating-point operators in a single design. In this paper, we discuss three advanced components recently added to our library: floating-point division (`fp_div`), floating-point

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06/$20.00 © 2006 IEEE

square root (`fp_sqrt`) and floating-point adder accumulation (`fp_acc`). These components all exhibit good tradeoffs between area, latency and throughput.

1.1 Related Work

Early implementations of floating point on FPGAs used non-standard formats, largely because implementing IEEE compliant single precision add and multiply was impractical [2, 3]. Some ideas from early papers incorporated into our library include turning normalization into a separate component [4] and adding signals to synchronize the flow of data through pipelines [5].

Floating-point division and square root are harder to implement due to the complexity of the algorithms. Dido et al. [6] calculate the inverse of the denominator based on a lookup table, followed by multiplying that inverse by the numerator. This method is only feasible for small dividers. Roesler and Nelson [7] implemented a Newton-Raphson based floating-point divider. However, it has long latency due to the iterative nature of the algorithm. The tradeoffs of designing a floating-point divider based on the higher radix SRT algorithm was studied by Wang and Nelson [8]. The IEEE double-precision floating-point divider described by Paschalakis and Lee [9] is a non-pipelined design that uses the simple radix-2 digit-recurrence algorithm. Therefore, it has small area but low throughput with a latency of 60 clock cycles for double-precision division. Underwood [10] discusses the feasibility of supporting IEEE double precision floating point on FPGAs. He presents add, multiply and divide components that are IEEE compliant. The divider is bit serial and exhibits long latency. square root. Previously published work on floating-point square root implementations [8, 9] uses the simple radix-2 digit-recurrence algorithm. The divider and square root presented in this paper differ from previously presented FPGA implementations. They are well suited to implementation on modern FPGAs because they use a mix of small table lookup and multipliers. They are also non-iterative and easily pipelined.

Floating-point accumulators and multiply-accumulators have been previously discussed and implemented [11, 12]. However, our design for the floating-point accumulator does not require stalling or compiler assistance [12], and does not rely on application characteristics to avoid data hazards [11]. Our design is well suited to be implemented on an FPGA.

The closest work to ours is the variable precision floating point library from Lyons [13]. Both libraries provide a variety of variable precision floating point components that are easily parameterizable. Both support conversion from different formats. Our library provides more support for pipelining and for exception handling. The divide and square root components in the Lyons library are SRT radix 4 and radix 2 respectively. These designs should exhibit longer latencies. In addition, their library does not include an accumulator component.

Users of our floating point library can make use of complementary research [14] focused on automatically optimizing the bitwidth of floating-point operands.

1.2 Floating-Point Library

Our parameterized floating-point library [1] consists of three types of components: format control, arithmetic operations, and format conversion. Format control includes modules `denorm` and `rnd_norm`. The first is used for denormalizing (introducing the implied integer bit that is required for computation) and the second is used for rounding and normalizing. Arithmetic operations include modules `fp_add`, `fp_sub`, `fp_mul`, `fp_div`, `fp_sqrt` and `fp_acc`, that are used for floating-point addition, subtraction, multiplication, division, square root and accumulation respectively. Format conversion includes modules `fix2float` and `float2fix`. The first is used to convert from fixed-point representation (both unsigned and signed) to floating-point representation and the second converts floating-point to fixed-point. Format conversion allows for the implementation of both fixed and floating-point computations within the same application.

Table 1 shows the name, function and latency (in clock cycles) of all the modules. Two of our advanced components `fp_div` and `fp_sqrt`, have latencies varying with the bitwidth of the floating-point format. We will see later that they have much longer latencies for wider bitwidth format than the other library modules due to their intrinsic complexity. The `fp_acc` component has a latency varying with both bitwidth and the number of operands. The reason for the longer latency of these three components is that we attempted to make all library modules have as similar a clock period as possible in order to achieve the highest throughput pipeline.

Our floating-point format is depicted in Fig. 1 along with two examples – IEEE 32-bit single-precision format and IEEE 64-bit double-precision format. Each hardware module of our library can be parameterized to accept any floating-point format with any bitwidth

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06/$20.00 © 2006 IEEE

Table 1: Floating-point Hardware Modules and Their Latency in Clock Cycles

Module	Function	Latency
`denorm`	Introduction of implied integer digit	0
`rnd_norm`	Normalizing and rounding	2
`fp_add/fp_sub`	Addition/Subtraction	4
`fp_mul`	Multiplication	3
`fp_div`	Division	variable
`fp_sqrt`	Square Root	variable
`fp_acc`	Accumulator	variable
`fix2float/fix2float_unsigned`	Unsigned/signed fixed-point to floating-point conversion	4/5
`float2fix/float2fix_signed`	Floating-point to unsigned/signed fixed-point conversion	4/5

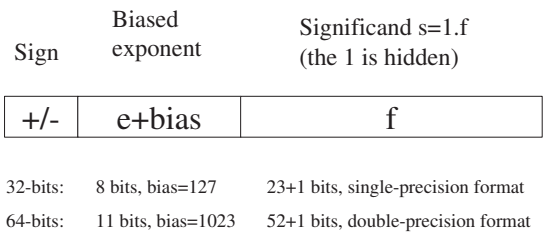

Figure 1: The ANSI/IEEE Standard Floating-Point Number Representation Formats

exponent and mantissa. All designs in our library are written in VHDL. The VHDL description of each module accepts the parameters *exp_bits* and *man_bits*. The *exp_bits* represents the bitwidth of the exponent and *man_bits* represents the bitwidth of the mantissa. These parameters are compile-time variables; they allow the bitwidth of each unit in the datapath to be parameterized, thus optimizing the amount and the structure of logic needed to implement a circuit. In other words, the designer has the flexibility to build a custom datapath by specifying the exact bitwidth of each datapath unit.

Fig. 2 shows a generic library component. Each

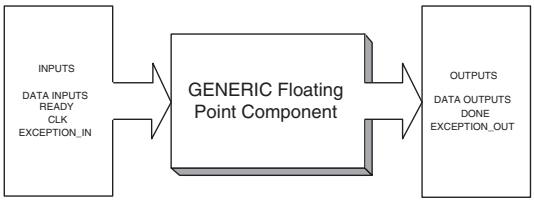

Figure 2: A Generic Library Component

component in our library has a READY signal and a DONE signal. When the READY signal is high, the component starts to process the input data which should be valid then or earlier. Upon completion, the component makes the output values available on the data outputs and sets the DONE signal to high. These handshaking signals allow our library components to be easily assembled into a pipelined datapath.

Our library also supports some error handling. Each module in our library has an EXCEPTION_IN signal and an EXCEPTION_OUT signal. An error fed into one component with the EXCEPTION_IN signal high or detected inside this component, is propagated to the EXCEPTION_OUT signal. In this manner, error signals are propagated through a pipeline along with their corresponding results.

In this paper we discuss the newest additions to our floating-point library: division, square root, and accumulation. These three advanced components are presented in Section 2. In Section 3, we describe the area, latency and throughput of a wide range of floating-point formats for these three components. Finally, discussion and conclusions are in Sections 4 and 5.

2 Advanced Floating-Point Components

Divide, square root, and accumulate are important operations in many high performance signal processing applications. We have implemented floating-point division, square root and accumulation. Both the division [15] and square root [16] algorithms are based on table lookup and Taylor series expansion, and use a combination of small table lookup and small multipliers to obtain the first few terms of the Taylor series. These algorithms are particularly well-suited for implementation on an FPGA with embedded RAM and embedded multipliers such as the Altera Stratix and Xilinx Virtex family devices. They are also non-iterative algorithms which makes them easy to fit into

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06/$20.00 © 2006 IEEE

the pipeline of a big design with other library modules without affecting the throughput of the whole design.

An accumulate component has been designed and implemented for our floating-point library that uses a buffering approach[17] to avoid data hazards. This approach uses a minimum of device resources, scales well with increasing mantissa size and operates at a frequency comparable to that of a single floating-point add. It is the only iterative component in our library.

2.1 Floating-Point Divider

The divider we built follows a previously published algorithm [15]. Assume a dividend X and a divisor Y, both $2m$ bit fixed-point numbers in the range of [1,2). They can be expanded in the form:

$$X = 1 + 2^{-1}x_1 + 2^{-2}x_2 + \cdots + 2^{-(2m-1)}x_{2m-1} \quad (1)$$
$$Y = 1 + 2^{-1}y_1 + 2^{-2}y_2 + \cdots + 2^{-(2m-1)}y_{2m-1} \quad (2)$$

where $x_i, y_i \in \{0, 1\}$. The divisor Y can be further decomposed into a higher order bit part Y_h and a lower order bit part Y_l, which are defined as:

$$Y_h = 1 + 2^{-1}y_1 + 2^{-2}y_2 + \cdots + 2^{-m}y_m \quad (3)$$
$$Y_l = 2^{-(m+1)}y_{m+1} + 2^{-(m+2)}y_{(m+2)} + \cdots + 2^{-(2m-1)}y_{2m-1} \quad (4)$$

The range of Y_h is between 1 and $Y_{hmax}(= 2 - 2^{-m})$, and the range of Y_l is between 0 and $Y_{lmax}(= 2^{-m} - 2^{-(2m-1)})$. To calculate X/Y using the Taylor series:

$$\frac{X}{Y} = \frac{X}{Y_h + Y_l} = \frac{X}{Y_h}(1 - \frac{Y_l}{Y_h} + \frac{Y_l^2}{Y_h^2} - \cdots) \approx X \times (Y_h - Y_l) \times \frac{1}{Y_h^2} \quad (5)$$

Since $Y_h > 2^m Y_l$, the maximum fractional error in equation (5) is less than 2^{-2m}, or 1/2 *ulp* [1]. Division based on equation (5) is also straightforward to implement in FPGA hardware. It requires only two multiplications and one table-lookup for $\frac{1}{Y_h^2}$. Fig. 3 shows a schematic representation of equation (5).

The BlockRAMs and embedded multipliers that the Xilinx Virtex-II FPGA provides are used in our implementation for table lookup and multiplication. Table lookup via BlockRAM takes only one clock cycle. The multipliers are pipelined multipliers and their latency varies with the size of the multiplier. Thus, the overall latency of the divider also varies with its size.

[1] *ulp* is an acronym for unit in the last place. The least significant bit of the fraction of a number is the last place

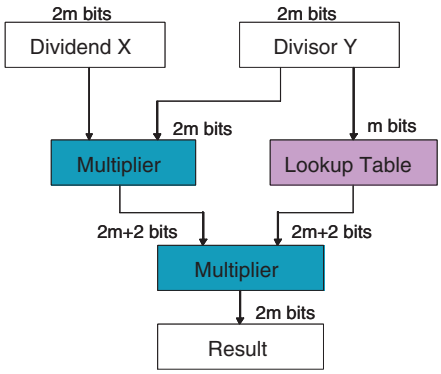

Figure 3: Table-Lookup Based Divider

Using this table-lookup based divider as the core, a floating-point divider can be built. Floating-point division can be implemented as:

$$\frac{(-1)^{s_1} \times m_1 \times 2^{e_1}}{(-1)^{s_2} \times m_2 \times 2^{e_2}} = (-1)^{s_1 \oplus s_2} \times (m_1/m_2) \times 2^{(e_1 - e_2)} \quad (6)$$

The sign of the quotient is the exclusive OR (XOR) of the signs of the input operands; the exponent of the quotient is the difference of exponents of the input operands; and the mantissa of the quotient is the quotient of mantissas of the input operands. The mantissa is obtained using the table-lookup based divider core shown in Fig. 3. The computation of sign, exponent, and mantissa can be implemented in parallel. Exception detection is implemented in parallel with the computation of the quotient mantissa.

2.2 Floating-Point Square Root

The square root we built is also based on a previously published algorithm [16]. It also uses table lookup and Taylor series expansion, but it is much more complicated than the divider. Fig. 4 shows the three steps to complete a square root: reduction, evaluation, and post processing. Let Y be an n bit fixed-point number in the range of [1,2) and let k be an integer such that

$$\boldsymbol{k = \lceil n/4 \rceil}$$

The first step is to reduce Y to a k bit number A $(-2^{-k} < A < 2^{-k})$ such that the square root of Y can be easily obtained from a function of $f(A)$. This $f(A)$ will be evaluated in the next step. A can be obtained as:

$$A = Y \times \hat{R} - 1 \quad (7)$$

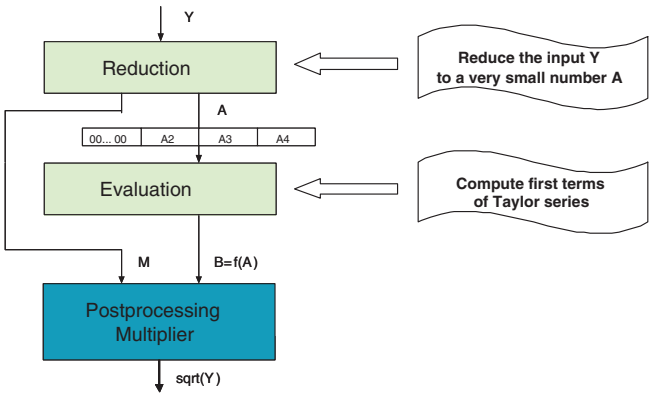

Figure 4: Table-Lookup Based Square Root

$$\hat{R} \quad = \quad 1/Y^{(k)} \tag{8}$$

where $Y^{(k)}$ is Y truncated to its k^{th} bit. A $4k$ bit number M is also generated:

$$M = 1/\sqrt{\hat{R}} \tag{9}$$

This M will be used in the post processing step to correct the final result due to the reduction step. Fig. 5 shows the data flow of this reduction step. Two table-lookups with k address bits each are required for $\hat{R}$ and M respectively and one $(k+1) \times 4k$ multiplier is required for A.

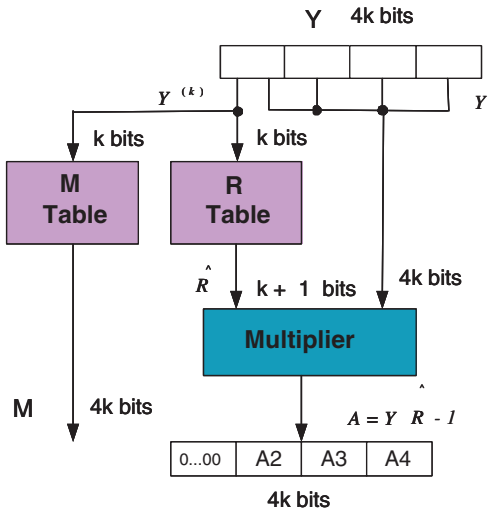

Figure 5: Reduction Step of Square Root

The second step is evaluation. Since $-2^{-k} < A < 2^{-k}$, A has the form:

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 + \cdots \tag{10}$$

where $z = 2^{-k}$ and $|A_i| = 2^k - 1$. Our goal is to compute the approximation $B = \sqrt{1+A}$. Using Taylor series expansion:

$$\begin{aligned} B &= \sqrt{1+A} \\ &\approx 1 + \tfrac{A}{2} - \tfrac{1}{8}A_2^2 z^4 - \tfrac{1}{4}A_2 A_3 z^5 + \tfrac{1}{16}A_2^3 z^6 \end{aligned} \tag{11}$$

This requires only two $k \times k$ multipliers and one $2k \times k$ multiplier as shown in Fig. 6.

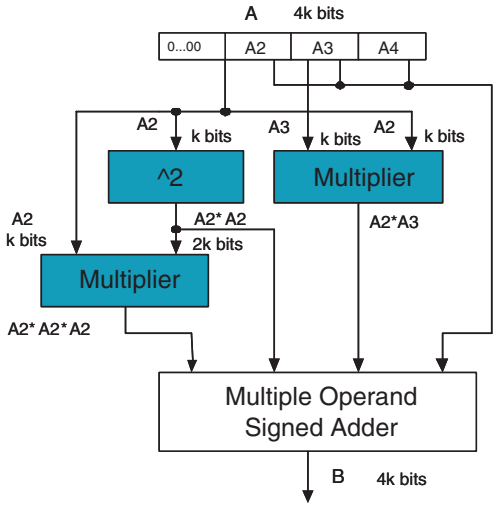

Figure 6: Evaluation Step of Square Root

At the end, we need to correct the result B by multiplying it by M from the reduction step. This requires one multiplier and is done in the post processing step. The overall error of all three steps is analyzed as 2.39×2^{-4k} [16].

Similar to division, lookup tables are built using BlockRAMs with one clock cycle latency. Multipliers are built using embedded multipliers which are pipelined with various latencies depending on their size. So the overall latency of the square root also varies with its size.

Using this table-lookup based square root as the core, a floating-point square root can be built. The sign of the square root and input operand should always be 0. Otherwise, an error is raised. The exponent of the square root is half the exponent of the input operand if the input exponent is even. If the input exponent is odd, the exponent of the square root is half the input exponent minus one and the mantissa of the square root needs to be multiplied by a constant

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06/$20.00 © 2006 IEEE

factor $\sqrt{2}$. We combine this constant multiplication in the computation of M in the reduction step. In other words, we have one extra table for $\sqrt{2}M$. Whether M or $\sqrt{2}M$ is selected depends on whether the input exponent is even or odd. The mantissa of the square root is the square root of the mantissa of the input operand, which can be obtained using the table-lookup based square root core shown in Fig. 4. Similar to other components, the exponent and mantissa are computed in parallel since they are independent. Exception handling is implemented in parallel as well.

2.3 Floating-Point Accumulator

The accumulator in our library is based on a floating-point adder. A buffer is inserted to ensure that no data hazards occur. A similar idea was used by Zhuo and Prasanna [17] to implement their *reduction circuit* for matrix vector multiplication. The general concept is as follows:

Assuming a new, valid input arrives every clock cycle, every two clock cycles there is enough data (i.e. two operands) to begin the pipelined floating-point add. Once the normalized output is produced, it can be buffered and re-introduced into the adder input chain in-between the times that new data is input into the add unit. The number of additions per cycle converges to one for large numbers of continuously valid new inputs and the maximum buffer size required is three words. In addition, none of the operands are dependent since they have traversed the entire pipeline and have been written back. These statements have been examined and proven correct [17].

The control logic behind the sets of conditions required are implemented as separate state machines, one controlling the insertion of new operands into the iterative addition chain and the other managing the feedback of old outputs back into the chain. The outputs of these state machines dictate the actions of the multiplexer and demultiplexer shown in Fig. 7, as well as control the registering of values. Overall latency information for the accumulator is presented in Table 2.

Component	Latency(Cycles)
Input mux	1
FP_add	4
Normalize	2
Buffering	1

Table 2: Table of Component Latencies

The accumulator differs from other library components in that normalization is implicit in the operation. The normalizer used is an updated version of the one described in [1]. Rather than use one large leading-zero counter, it breaks the mantissa into two sections, counts the number of leading zeroes in each section and adds them together depending on some additional control logic. This method of dividing up the leading-zero counter provides faster and smaller normalize implementations [11]. We plan to use this approach to improve the normalize component in the library.

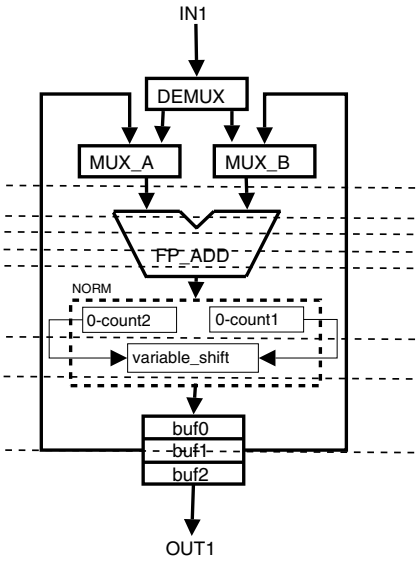

Figure 7: Accumulate Decomposed Into Components

Exception handling is provided via the EXCEPTION_IN and EXCEPTION_OUT lines. An exception input to the accumulator causes the related operand to be set to zero and the accumulation continues. An output exception is asserted when the accumulation is complete and the calculated sum (not including the erroneous input) is driven on the output lines. Overflow and underflow exceptions are generated internally by the fp_add unit. These cause the output to wrap to zero and continue accumulating. In this event an exception is also asserted upon completion.

3 Experimental Results

3.1 Experimental Setup

All designs presented in this paper are implemented on an Atlanta board from Mercury Computer Systems, Inc. Fig. 8 shows the module architecture of this board. This board integrates an FPGA with Pow-

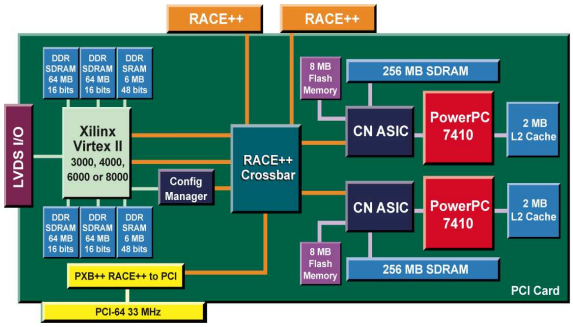

Figure 8: Architecture of the Mercury Atlanta Board

erPC G4 microprocessors in the RACEway switch fabric architecture. Some of the main features of the Atlanta board are: one Xilinx Virtex-II XC2V6000 FPGA compute node at a speed grade of -5; 12MB of DDR SRAM and 256MB of DDR SDRAM both running at 133MHz; 52 pairs of LVDS I/O lines; and a dual-processor PCI module with two PowerPCs. Each PowerPC compute node consists of a 500 MHz PowerPC 7410 microprocessor. In this work we only make use of the FPGA.

All our designs are written in VHDL and synthesized using Synplify Pro 8.0. The bitstream is generated using Xilinx ISE 6.3i and downloaded to the Virtex-II XC2V6000 FPGA on the Atlanta board. This section presents experimental results for each of the three advanced modules (*fp_div*, *fp_sqrt* and *fp_acc*) for a wide range of floating-point formats including IEEE single-precision and double-precision formats. All the advanced hardware modules have been tested both in simulation and in hardware. For simulation, we use a testbench that tests a combination of random numbers and corner cases for various bitwdith floating-point formats. For testing on the reconfigurable board, we send the same set of numbers through the PCI bus to the FPGA board, do the computation on the board, then read them back to the host PC.

3.2 Experimental Results for Floating-Point Divider and Square Root

Tables 3 and 4 show the area, latency and throughput of several different floating-point formats including IEEE single-precision format for division and both IEEE single-precision and double-precision format for square root. The format is represented by the total number of bits, the number of exponent bits followed by the number of mantissa bits are shown in parentheses. The sign bit is assumed. The fifth column of both tables represents IEEE single-precision format $32(8, 23)$. The last column of Table 4 is the IEEE double-precision format $64(11, 52)$. The quantities for the area of each design in both tables are evaluated using three metrics - number of slices, number of on-board BlockRAMs and number of 18×18 embedded multipliers. The Xilinx Virtex-II XC2V6000 FPGA has a total number of 33792 slices, 144 on-board BlockRAMS, and 144 embedded multipliers.

Our results show that both the area and the latency of our floating-point divider and square root are small compared to most other divider and square root implementations. For an IEEE single-precision format divider, it takes 14 clock cycles to generate the final result with a $7.7ns$ clock period resulting in a total latency of $108ns$. Note that non-iterative algorithms are employed for both our divider and square root compared to most other divider and square root implementations. The non-iterative nature of the algorithm allows them be fully pipelined and thus the throughput is one result per clock cycle. This allows them to easily fit into the pipeline of a big design with other library modules without decreasing the throughput of the whole design. The throughput of our IEEE single-precision format divider is as high as 129 million results per second. Meanwhile, this design only takes 1% of the slices, 4% of the BlockRAMs, and 5% of the embedded multipliers on the FPGA chip. Similarly, our floating-point square root shows a good tradeoff between area, latency and throughput.

It is an obvious observation from both tables that the wider the floating-point bitwidth, the larger and slower the circuit. For wider designs, not only does the clock period increase, but also the number of clock cycles to generate the final result. Our goal is to keep the clock period relatively constant over a wide range of bitwidths and formats. Therefore we have to add more pipeline stages to wider bitwidths as a compromise. This results in a steadily increasing latency with increasing bitwidth.

The largest floating-point components that we can implement is limited by the number of on-board BlockRAMs. As we can see from both tables, the number of BlockRAMs required increases at a much faster rate compared to the number of slices and the number of embedded multipliers as bitwidth increases. This is because the sizes of the tables (in Fig. 3) for the divider and (Fig. 5) square root increase exponentially while other parts of the design increases polynomially with the increase in bitwidth. Fig. 3 shows that divider requires one table with the size of $2^m \times (2m+2)$.

Table 3: Area, Latency and Throughput for Floating-Point Division

Floating Point Format	8(2,5)	16(4,11)	24(6,17)	32(8,23)	40(10,29)
number of slices	66 (1%)	115 (1%)	281 (1%)	361 (1%)	617 (1%)
number of BlockRAM	1 (1%)	1 (1%)	1 (1%)	7 (4%)	62 (43%)
number of 18×18 embedded multiplier	2 (1%)	2 (1%)	8 (5%)	8 (5%)	8 (5%)
clock period (ns)	4.9	6.8	7.8	7.7	8.0
maximum frequency (MHz)	202	146	129	129	125
number of clock cycles to generate final results	10	10	14	14	14
latency(ns) = clock $\times$ number of clock cycles	49	68	109	108	112
throughput (million results per second)	202	146	129	129	125

Table 4: Area, Latency and Throughput for Floating-Point Square Root

Floating Point Format	8(2,5)	16(4,11)	24(6,17)	32(8,23)	48(9,38)	64(11,52)
number of slices	85 (1%)	172 (1%)	308 (1%)	351 (1%)	779 (2%)	1572 (4%)
number of BlockRAM	3 (2%)	3 (2%)	3 (2%)	3 (2%)	13 (9%)	116 (80%)
number of 18×18 embedded multiplier	4 (2%)	7 (4%)	9 (6%)	9 (6%)	16 (11%)	24 (16%)
clock period (ns)	6.1	7.2	7.8	8.0	8.8	9.7
maximum frequency (MHz)	165	139	129	125	114	103
number of clock cycles to generate final results	9	12	13	13	16	17
latency(ns) = clock $\times$ number of clock cycles	55	86	101	104	140	165
throughput (million results per second)	165	139	129	125	114	103

The address of this table is m-bits wide, where m is half the width of the mantissa bitwidth. Fig. 5 shows square root has more tables but the address of each table is k-bits wide, which is only 1/4 of the mantissa bitwidth. Therefore, the size of tables for the square root increases much more slowly than that of the divider. For small bitwidths, square root requires more BlockRAMs than the divider, while for large bitwidths, it requires fewer. The largest floating-point square root we can design is IEEE double-precision 64(11,52), which requires about 80% (116 out of 144) of on-board BlockRAMs for this chip. The largest floating-point divider we can implement on the Virtex-II 6000 is the 40(10,29) format. This format requires about 43% (62 out of 144) of on-board BlockRAMs. Since the table size for the divider more than doubles when the width of the address increases just by one bit, the next larger design will result in insufficient on-board BlockRAMs on this chip. We are investigating using a similar algorithm as that used for square root to implement division.

3.3 Experimental Results for Floating-Point Accumulator

The number of clock cycles from the de-assertion of the READY line (signaling the end of valid input), to the assertion of the DONE line by the accumulator varies between nine cycles for only two inputs, to a maximum of thirty five cycles for a sufficiently large number of inputs. This is for an eight cycle pipeline length. Increasing the pipeline length by one increases the worst case delay to more than forty cycles.

The rate at which the overall cycle time as well as the latency increases with respect to the number of operands is shown in Fig. 9. For an even number of operands, the worst case latency is thirty two cycles. For an odd number it is thirty five cycles.

The results presented in Table 5 are taken from post place and route reports using the Xilinx ISE 6.3i toolset. The floating point representation differs from that above in that 3 guard bits are used to maintain accuracy. As can be seen, the slice count increases approximately linearly with respect to bitwidth. Frequency decreases by 23 percent when going from single-precision to double-precision. Unlike `fp_div` and `fp_sqrt`, this component does not utilize BlockRAMs or embedded multipliers and its overall latency depends on the number of operands.

Although a single floating-point add based on the `fp_add` component takes up fewer slices than an equivalent bitwidth accumulator, it operates at a similar frequency. A single-precision add (with the same num-

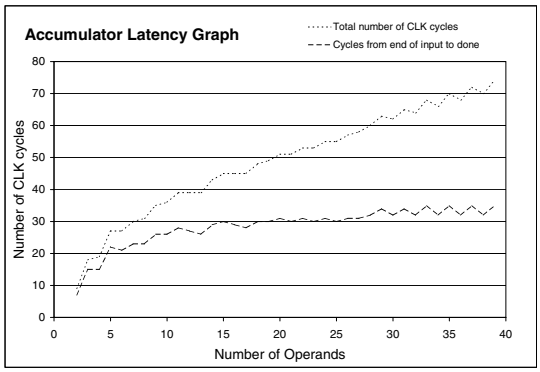

Figure 9: Graph of Accumulator Delay

Bitwidth (E,M,Grd)	24 6,17,3	32 8,23,3	47 9,37,3	64 11,52,3
Slices	431	558	843	1145
Freq. (MHz)	149	140	123	108
Period (ns)	6.7	7.1	8.1	9.25

Table 5: Table of Accumulator Frequency vs. Size

ber of guard bits) operates at 136 MHz, while a double precision version operates at 126 MHz. Thus the accumulator speed is within 15% of the floating-point add for the sizes discussed here.

4 Discussion

Many signal processing algorithms can be accelerated using reconfigurable hardware. To achieve a good speedup compared to running software on a general purpose processor, fine-grained control over the bitwidth of each component in the datapath is desired. This goal can be achieved by using our variable precision floating-point library. To demonstrate the division implementation, we incorporate it into our previous implementation of the K-means clustering algorithm applied to multispectral satellite images [1]. With the lack of floating-point divider, the mean updating step in each iteration of the K-means algorithm has to be moved to the host computer for calculation. The new means calculated on the host then have to be moved back to FPGA board for next iteration of the algorithm. This dramatically decreases the speedup of the hardware implementation. With our new `fp_div` module, we are able to implement the mean updating step in FPGA hardware. This greatly reduces the communication overhead between host and FPGA board and further accelerates the runtime. We are also investigating using the new `fp_acc` component in this implementation.

Note that on-board BlockRAMs are used for all our tables in this work. Our philosophy is to reserve LUTs for other components, such as adders. Hence our design is limited by the size of on-board BlockRAMs. In Table 3, the largest divider 40(10,29) uses 43% of BlockRAMs but only 1% of slices. Larger dividers could be implemented by using LUTs as distributed memory. We are planning to investigate this in future designs to achieve wider bitwidths.

The method of accumulation used in the `fp_acc` component scales well in terms of speed and size as bitwidth increases. This can clearly be seen in Table 5. Its greatest drawback is that the delay from the end of input to the assertion of the DONE signal increases rapidly with respect to pipeline length. Thus optimal use of this accumulator is made when utilizing short pipelines or over a large number of accumulations. Comparing the accumulator with an implementation using a stalling technique and a four-stage adder, this point occurs with an accumulation of eight floating-point numbers.

Rounding support is currently provided via a parameterizable number of guard bits. The library rounding module can then be used post-accumulation to provide a more accurate final sum. If a greater degree of rounding accuracy is required, the normalize in the accumulator can be replaced with the full `round_norm` library component to provide round-to-nearest functionality. This is easily accomplished since the `round_norm` and the accumulator normalize both take two cycles and use similar inputs and outputs.

5 Conclusions

Three advanced floating-point components - division (`fp_div`), square root (`fp_sqrt`) and accumulation (`fp_acc`) have recently been added to our variable precision floating-point library. All three of these advanced components as well as all other hardware modules in our floating-point library are fully parameterized and fully pipelined. Several different floating-point formats, including IEEE single-precision and double-precision formats for these components are discussed. Our results show that these components exhibit a good tradeoff between area, latency and throughput, and are easily

pipelined. Our library supports the creation of custom format floating-point datapaths, as well as hybrid fixed and floating-point implementations using these components. All the components are available as part of the variable precision floating point library from:
`www.ece.neu.edu/groups/rcl/projects/floatingpoint/`.

Acknowledgements

This research was funded in part by Mercury Computers and by the NSF ERC Center CenSSIS.

References

[1] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *12th International Conference on Field Programmable Logic*, pp. 657–666, Sept. 2002.

[2] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 365–367, Sept. 1994.

[3] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 155–162, April 1995.

[4] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 107–116, April 1996.

[5] C. S. Gloster, Jr. and I. Sahin, "Floating-Point Modules Targeted for Use with RC Compilation Tools," in *Earth Science Technology Conference*, Aug. 2001.

[6] J. Dido, N. Geraudie, *et al.*, "A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs," in *International Symposium on Field Programmable Gate Arrays*, pp. 50–55, Feb. 2002.

[7] E. Roesler and B. E. Nelson, "Novel optimizations for hardware floating-point units in a modern FPGA architecture," in *12th International Conference on Field-Programmable Logic*, pp. 637–646, 2002.

[8] X. Wang and B. E. Nelson, "Tradeoffs of designing floating-point division and square root on Virtex FPGAs," in *11th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 195–203, April 2003.

[9] S. Paschalakis and P. Lee, "Double precision floating-point arithmetic on FPGAs," in *IEEE International Conference on Field-Programmable Technology*, pp. 352–358, Dec. 2003.

[10] K. D. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *12th International Symposium on Field Programmable Gate Arrays*, pp. 171–180, Feb. 2004.

[11] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *13th International Symposium on Field-Programmable Gate Arrays*, Feb. 2005.

[12] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," in *IEEE Transactions on Computers*, vol. 49, pp. 208–218, March 2000.

[13] "A VHDL library of parametrisable floating-point and LNS operators for FPGA." http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/.

[14] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung, "Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs," in *12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 79–88, April 2004.

[15] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn, "Fast division algorithm with a small lookup table," in *33rd Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1465–1468, May 1999.

[16] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Transactions on Computers*, vol. 49, pp. 628–637, July 2000.

[17] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing scalable FPGA-based reduction circuits using pipelined floating-point cores," in *19th International Parallel and Distributed Processing Symposium*, April 2005.